NUMA-AWARE DATA MANAGEMENT FOR NEUTRON CROSS SECTION DATA IN CONTINUOUS ENERGY MONTE CARLO NEUTRON TRANSPORT SIMULATION

Nicolas Denoyelle,[1] John Tramm,[1] Kazutomo Yoshii,[1] Swann Perarnau,[1] and Pete Beckman[1]
[1]Argonne National Laboratory
9700 S. Cass Avenue, Lemont, IL, USA

{ndenoyelle,jtramm,kazutomo,swann}@anl.gov,~beckman@mcs.anl.gov

ABSTRACT

The calculation of macroscopic neutron cross-sections is a fundamental part of the continuous-energy Monte Carlo (MC) neutron transport algorithm. MC simulations of full nuclear reactor cores are computationally expensive, making high-accuracy simulations impractical for most routine reactor analysis tasks because of their long time to solution. Thus, preparation of MC simulation algorithms for next generation supercomputers is extremely important as improvements in computational performance and efficiency will directly translate into improvements in achievable simulation accuracy. Due to the stochastic nature of the MC algorithm, cross-section data tables are accessed in a highly randomized manner, resulting in frequent cache misses and latency-bound memory accesses. Furthermore, contemporary and next generation non-uniform memory access (NUMA) computer architectures, featuring very high latencies and less cache space per core, will exacerbate this behaviour. The absence of a topology-aware allocation strategy in existing high-performance computing (HPC) programming models is a major source of performance problems in NUMA systems. Thus, to improve performance of the MC simulation algorithm, we propose a topology-aware data allocation strategies that allow full control over the location of data structures within a memory hierarchy. A new memory management library, known as AML, has recently been created to facilitate this mapping. To evaluate the usefulness of AML in the context of MC reactor simulations, we have converted two existing MC transport cross-section lookup "proxy-applications" (XSBench and RSBench) to utilize the AML allocation library. In this study, we use these proxy-applications to test several continuous-energy cross-section data lookup strategies (the nuclide grid, unionized grid, logarithmic hash grid, and multipole methods) with a number of AML allocation schemes on a variety of node architectures. We find that the AML library speeds up cross-section lookup performance up to 2x on current generation hardware (e.g., a dual-socket Skylake-based NUMA system) as compared with naive allocation. These exciting results also show a path forward for efficient performance on next-generation exascale supercomputer designs that feature even more complex NUMA memory hierarchies.

1. Introduction

In this study we investigate the impact of non-uniform memory access (NUMA) effects on the performance of the Monte Carlo (MC) neutron transport method in the context of full-core nuclear reactor eigenvalue simulations. Additionally, we compare the effectiveness of different methods for mitigating these impacts. One such method is to use the AML library to replicate latency-sensitive structures critical to the MC transport such that each thread on a node can access the data from a location with minimal latency. In this section, we briefly introduce the basics of MC neutron transport and the computational challenges facing it. We then introduce AML and explain how it can be used to potentially boost the performance of the MC transport method on NUMA architectures.

1.1. Monte Carlo Neutron Transport

The purpose of a Monte Carlo neutron transport reactor simulation is to gain useful data about the distribution and generation rates of neutrons within a nuclear reactor. In order to achieve this goal, a large number of neutron lifetimes are simulated by tracking the path and interactions of a neutron through the reactor from its birth in a fission event to its escape or absorption, the latter possibly resulting in another fission event.

Each neutron in the simulation is described by three primary factors: its spatial location within a reactor's geometry, its speed (i.e., energy), and its direction. At each stage of the transport calculation, a determination must be made as to what the particle will do next. For example, possible outcomes include uninterrupted continuation of free flight, collision with another atom, or fission with a fissile material. The determination of which event occurs is based on a random sampling of a probability distribution that is determined by empirical material cross-section data stored in main memory. This data, called *macroscopic neutron cross-section data*, represents the probability that a neutron of a particular speed (energy) will undergo some particular interaction when it is inside a given type of material.

All materials in a reactor are composed of multiple nuclides. Each nuclide within a material has its own independent *microscopic* cross-section data. Therefore, in order to determine the overall probability of a reaction occurring when a neutron is traveling through a material, macroscopic cross-sections must be assembled from the underlying isotopic data sets, as

$$\Sigma_c(\text{Material, Energy}) = \sum_{\text{Nuclide } n}^{\text{Material}} \rho_n \times \sigma_{c,n}(\text{Energy}), \tag{1}$$

where Σ_c is the macroscopic cross-section for reaction channel c, ρ_n is nuclide n's density within the material, and $\sigma_{c,n}$ is nuclide n's microscopic cross-section for reaction channel c.

Besides being used to assemble macroscopic cross-sections, microscopic cross-sections are used to generate cumulative distribution functions that can be sampled to determine the specific nuclide a neutron interacts with and by which channel.

As a neutron travels through a reactor, from birth at high energy from fission to death via absorption or escape from the reactor, its slowing-down process and interactions will all be governed by this cross-section data. Since neutrons exist in the reactor at energies spanning over 10 orders of magnitude and since slowing down can randomly cause neutrons to lose up to half their energy in one scattering event, cross-section data will necessarily be accessed in a manner that exhibits low spatial and temporal locality. Thus, efficient algorithms for organizing cross-section data and accessing it are of paramount importance to the overall MC transport method. In fact, some high-fidelity, full-core reactor simulations spend up to 85% of their overall runtime [1] just looking up and assembling cross-section data.

1.2. Monte Carlo Miniapps: XSBench and RSBench

In this analysis, we investigate the impact of NUMA effects on the performance of the Monte Carlo neutron transport simulations and compare different methods for mitigating this impact. Rather than performing our analysis on a full MC transport application such as OpenMC[2], we instead use several MC "miniapps" that serve as stand-ins for the full applications.

The XSBench [1] and RSBench [3] miniapps represent different forms of the most computationally expensive kernel of a MC particle transport simulation in the context of nuclear reactor core simulations. These miniapps recreate the essential computational conditions and tasks of fully featured MC transport codes when simulating full-core 3D reactor problems with depleted fuel featuring hundreds of nuclides, without the additional complexity of the full application. This approach provides a simpler and more accessible platform for isolating where both hardware and software bottlenecks degrade the performance of cross-section lookup algorithms. XSBench and RSBench are therefore ideal for this analysis rather than using the full OpenMC application because they can be rapidly edited to use the AML allocation scheme and then can be deployed on a wide variety of hardware architectures without having to deal with the added complexity (and software dependency chain) of a full application.

XSBench and RSBench have been used extensively as stand-ins for full Monte Carlo transport applications. XSBench has even been empirically validated against a full particle transport application OpenMC on CPU architectures to show that the miniapp produces hardware performance conditions similar to OpenMC, as measured by key performance counter rates (e.g., cache miss rates, FLOP rates, TLB miss rates) [1].

Collectively, XSBench and RSBench can perform the following types of continuous-energy cross-section lookups:

- **Nuclide Grid.** This is the "naive" lookup method for storing and accessing cross-section data, as discussed in [4]. In this method, pointwise cross-section data is stored in table format at separate energy levels for all nuclides. This method is highly inefficient because each macroscopic cross-section lookup will require that hundreds of binary searches be performed—one search for each nuclide in the material.
- **Unionized Grid.** This method, detailed in [4], adds an "acceleration" structure to the lookup algorithm that requires only one binary search be performed for each macroscopic lookup. This has the effect of greatly increasing the speed of the lookup, although at the cost of a significantly increased (over 10x) memory footprint for cross-section data.

- **Logarithmic "Hash" Grid.** This method, detailed in [5], also adds a small acceleration structure to the algorithm. The "hash" grid is similar to the unionized method, although the number of gridpoints is capped to a much smaller size, and only an approximate index is given into the underlying nuclide grid for each nuclide. The net result is that overall lookup speed is competitive with the unionized grid method but with a much lower memory requirement.

- **Multipole.** Unlike all the other pointwise table-based lookup methods, the multipole method stores the underlying resonance parameters so that cross-section data can be built on the fly. This cuts the data requirements significantly, using at most a few tens of megabytes, although at the cost of significantly more floating-point work being required. The multipole method is fully detailed in [6].

Each of these lookup methods is important to study because they can each perform differently based on which particular hardware architecture is used.

1.3. Monte Carlo Performance Challenges

In Monte Carlo, cross-section data is frequently read from in a manner that exhibits low spatial and temporal locality. The MC transport method is therefore highly taxing on the memory subsystems of a computer, causing low cache efficiency and typically resulting in latency-bound performance. When running in parallel on NUMA architectures, these latency bottlenecks can potentially become greatly exacerbated, significantly reducing performance.

When the application uses one process per compute node, data structures are allocated on a single NUMA node. Subsequent concurrent access from the whole node to these structures will create memory contention and remote memory access. One way to optimize the latency of memory access is to spawn one MPI process per socket of the processor. In this scenario, a copy of *all* application data will be allocated on the memory close to processes threads, therefore limiting memory contention and improving locality. However, the application may not be able to afford the memory overhead of such an extensive duplication. In the case of Monte Carlo transport applications, cross-section data typically has a footprint in the range of tens of megabytes (for multipole [3]) up to 6 GB (when using pointwise data with a unionized lookup grid [1]). However, high-fidelity full-core simulations featuring fine-grained burnup region meshes can require up to 1 TB of tally data storage [7] to be replicated across all MPI ranks. Thus, launching a single MPI rank per NUMA subdomain is often impractical; instead, a single MPI rank per node is often preferable. If physical domain decomposition is used by the application, then the problem of tally data can be mitigated; but full applications such as OpenMC do not currently use this technique because of the great increase in code complexity that domain decomposition would require.

With these restrictions in mind, one way to mitigate the data overhead is to replicate only the latency-sensitive data structures (i.e., cross-section data) and provide threads with the closest replica [8]. However, managing data locality at this granularity is a concern better suited for a memory library, especially considering exascale computer architectures that feature deeper and more complex memory hierarchies.

1.4. Using AML Memory Library for Fine-Grained Management of MC Data

Similarly to MC particle transport, many memory-bound applications are facing new challenges as computer architectures evolve toward deep and highly heterogeneous memory hierarchies. Most large applications greatly outlive the machine's lifespan. Tuning these applications for the new capabilities with every new machine delivery represents a significant expense. It motivates the design of portable performance abstractions that contain the area of machine-specific code to engineer in order to support next-generation computing systems.

AML is a memory management library [*] written in C. It operates at the granularity of a single image system and is meant to be embedded in scientific or runtime libraries deploying efforts to optimize their data management [9]. Its purpose is to lay the foundations for designing portable performance optimizations such as the one showcased in this paper. The library is organized around a set of base abstractions for mapping data to memory, moving data around and describing data organization. It allows foreign implementations of its abstractions to increase the whole ecosystem's capabilities. For instance, one can provide different ways to map data to memory and different ways to move data.

This work focuses on the specific data-mapping abstraction in AML for implementing data replication. Aforementioned AML abstraction is named *area*. Areas represent places on a compute node where data can be mapped. Its implementation is bound to device drivers, for example CUDA for NVIDIA GPGPUs and libnuma for NUMA host processors. Areas provide a means to explicitly declare locations where data can belong and to allocate these places with a consistent interface.

In this study we implemented an area embedding topology information for NUMA machines. We further leveraged this type of area to build a topology-aware replication layer. We then used this layer inside XSBench and RSBench to replicate locality-sensitive data and perform local memory access. As a result, the XSBench miniapp performance was improved by up to one order of magnitude. Plugging in AML capabilities adds only 4% new code to the application. This addition yields a performance consistently equal to or better than low-effort optimizations of the same nature. Moreover, this new feature does not require tweaking the number of processes according to the system topology or embedding machine-specific code inside the application.

2. Contribution

2.1. Topology-Aware Areas and Data Replication

Topology-aware areas and data replication are implemented on top of the software package hwloc [10]. The package provides a portable abstraction layer for machines topology detection and query. `hwloc_area` adds topology information to the default `linux_area`. `linux_area` is an area based on linux `mmap` and libnuma `mbind` to map data on NUMA CPU memories. `hwloc_area` interfaces with the NUMA distances matrix retrieved from hwloc. Provided that application threads are bound, AML is then able to identify a thread location and pick the closest area where to map data.

[*] AML repository: https://xgitlab.cels.anl.gov/argo/aml

```
struct aml_replicaset * replicaset;

// Create replicaset areas and allocate space on areas.
aml_replicaset_hwloc_data_create(&replicaset, data_size,
// Track threads position at the processing unit granularity.
HWLOC_OBJ_PU,
// Create Areas for NUMANODEs
HWLOC_OBJ_NUMANODE,
// Subset NUMANODEs relatively to their OS defined latency to PUs.
HWLOC_DISTANCES_KIND_MEANS_LATENCY | HWLOC_DISTANCES_KIND_FROM_OS,
// Pick NUMANODEs with the smallest latencies.
aml_replicaset_hwloc_lt);

// Replicate data of size data_size.
aml_replicaset_init(replicaset, application_data, data_size);

// Get replica on NUMANode with least OS latency relatively to current
// thread binding.
void * close_data = aml_replicaset_hwloc_get(replicaset);
```

Figure 1. AML replicaset interface.

AML replicaset abstraction is designed on top of these building blocks. It consists of a set of areas and pointers to replicas allocated with the same areas. AML replicaset automatically creates area instances on a subset of the system memories based on a performance criterion. For instance, on an Knights Landing (KNL) system, configured in *SNC-4* and *flat* modes, AML will detect the four compute quadrants and their associated MCDRAM (high-bandwidth memory) and DRAM (regular memory). If the data needs to be replicated on high-bandwidth memories, AML will spawn one area per quadrant in MCDRAM. Then, on future queries for a replica in the replicaset, AML will identify from which quadrant each query has been made and will pick the data in the area attached to it.

From these capabilities stems a naturally simple interface shown in Figure 1. Using this interface requires few additions to existing codes. The resulting code is not machine specific; it will work on any machine that can be meaningfully described by hwloc abstraction. A minor tweak has been done for the KNL processor. Indeed, on this machine, the operating system defines MCDRAM distances to be very high in order to prevent the operating system from using it as the default memory and keep it free for applications. Therefore, on the KNL machine, NUMANODEs are picked with the greatest area distance instead of the smallest one. This can be fixed by providing hwloc with real performance values.

2.2. Modifications to XSBench and RSBench Code Base

We leverage replicaset abstraction to replicate XSBench and RSBench locality sensitive data structures. This is achieved by (1) adapting the application to ease the byte per byte copy of critical data, and (2) plugging AML data replication and locality-aware replica access capabilities inside the application.

Table I. Summary of code changes from original code (*master*), to optimized version (*aml-replicaset*) by a version with modified data structures (*no-aml*). Repository available at https://xgitlab.cels.anl.gov/argo/applications/XSBench

src branch	dst branch	src lines	unchanged	inserted	deleted
master	master	3822.00	100.00%	0.00%	0.00%
master	no-aml	3822.00	92.18%	11.80%	7.82%
no-aml	aml-replicaset	3974.00	99.97%	4.25%	0.03%
master	aml-replicaset	3822.00	92.23%	16.14%	7.77%

Briefly, XSBench data structures consist of a contiguous array of nuclide grid points and an array of energy points with a pseudo-random index in the nuclide grid. RSBench is slightly more complicated. It has two main data structures containing most of the application footprint. In either case, sparse arrays have been flattened to dense arrays. When possible, structures have been packed into a single one to avoid multiple allocations and replicasets. When modifying these structures, we tried to remain as conservative as possible. Most of the structures fields remained intact, and zero-overhead macro accessors have been provided otherwise. All the changes have been checked against the application built-in validation. The numerical results produced after modification remain correct.

With these modifications, adding AML support essentially consists of associating data pointers with a replicaset and providing threads with replica instead of the usual data structure. Changes from the `master` branch to the `no-aml` branch reshape the data structures to enable further use of the AML replication layer. They represent the most significant modifications applied to the application. Code changes are summarized in Table I. We note that 92% of the original code has been kept, while about 12% additions have been incorporated. The `aml-replicaset` branch contains the optional additions from `no-aml` branch, adding support for latency-sensitive data replication and locality-aware access. Provided that the code is shaped for such a feature, adding AML support adds only 4% new code to the code base.

2.3. Experimental Results on Performance and Portability

For XSBench and RSBench applications, allocation of sensitive data structures has been set with several strategies.

- The **Default** allocation strategy – This maps a single data copy in memory with `malloc`. A single MPI process is used for the whole machine. Default and other strategies are run several times with different numbers of threads, and the results with the optimal count are saved.
- **Interleave** allocation strategy – this further applies an interleave policy to bind data allocated with `malloc` across all memories.
- **AML replicaset** allocation strategy – This uses AML to allocate space on fast memories of the machine and replicate data there. In this configuration, each thread of the single MPI process requests a replica of data prior to initiating computations. AML looks for the thread position on

Table II. Systems specifications

family	model	#cores	frequency	#sockets	#NUMA	LLC size
haswell	Xeon E7-8867 v3	64	2.5 (GHz)	4	4	45 (MB)
skylake	Xeon Platinum 8180M	56	2.5 (GHz)	2	2	39 (MB)
epyc	EPYC 7601	64	2.2 (GHz)	2	8	8192 (KB)
knl	Xeon Phi 7250	68	1.4 (GHz)	(SNC-4) 1	8	1024 (KB)

the machine and provides the closest replica of the application data. The overhead of replication is not measured in this paper. From the gains described below, however, one can see that the overhead is likely to be negligible.

- **MPI** version – This is a best-effort attempt at improving data locality without modifying the original code. MPI version spawns one process per socket and uses the default allocation strategy.

We compared data management strategies for XSBench *hash*, *unionized*, and *nuclide* grids resolutions and RSBench *multipole* method. Applications were compiled with gcc (4.8.3) and `-O3 -march=native` optimization flags on the target machines. They have been run on a total of 4 multisocket systems, summarized in Table II.

RSBench data structures are small enough to fit the last-level cache (LLC) on Haswell and Broadwell machines and to fit the 32 distributed L2 cache tiles on the KNL system. Therefore, data access occurs mainly in the cache, and in-memory locality does not significantly impact performance. Figure 2 shows for RSBench multipole method with similar performance on Haswell, Skylake, and KNL for all allocation strategies. On the Epyc system with a tiny last-level cache, the allocation strategy does impact performance because the dataset is larger than the cache size. The remaining results focus on XSBench, which has a larger memory footprint.

Data allocated with the default allocation strategy will live in the memory close to the thread initializing data, in other words, a single memory. As a result, the application suffers from remote memory access and contention. Figure 2 shows the performance of this allocation strategy for several resolution methods on several machines. The default allocation strategy stands as the worst case in all scenarios, showing that memory management is necessary in order to get performance. Achievable improvements compared with this strategy are as high as one order of magnitude (*cf.* hash and unionized grids with AML_replicaset strategy).

Compared with the default policy, the interleave allocation strategy relieves the contention and averages remote access for all threads. Consequently, the performance improves with all input grids. On Haswell, Skylake, and Epyc, the MPI version outperforms the default and interleave strategies. In fact, on the Haswell and Skylake systems, the MPI strategy is nearly equivalent to the AML_replicaset strategy. It does bind one process per NUMA place, and the default allocation strategy applied in this configuration will lead to having data close to the processing units. The MPI version reaches the best performance on these two machines, but it comes with several downsides.

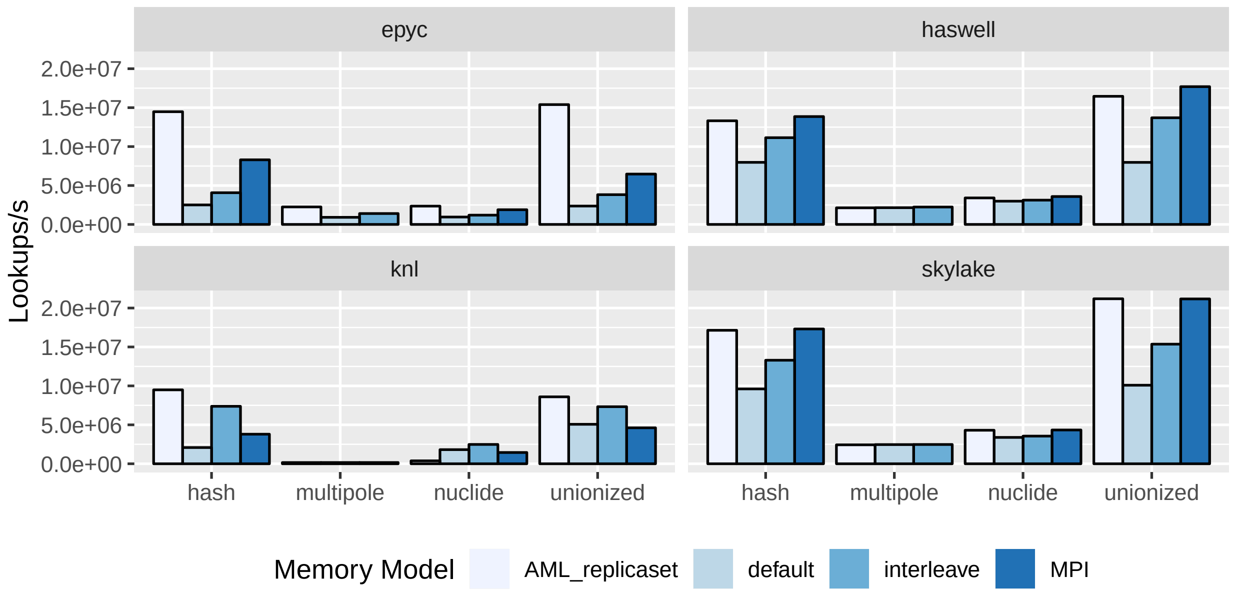

Figure 2. Comparison of memory allocation strategies performance in OpenMC miniapplications, for several resolution methods and several computing systems.

First, the MPI version will replicate not only latency-sensitive data but all application data. If a large piece of it does not need replication, it may become a concern. In particular, XSBench cross-section material data spans several gigabytes and may extend up to 1 TB for the full core reactor simulation. Such a scale is impractical to replicate for each MPI process. Replicating only data on which the computations focus would, however, fit the system memory. Second, the MPI version requires that the application user have knowledge of the machine topology in order to set the number of MPI processes and bind them to the good resources. Indeed, although the strategy of having one process per socket was a good fit on Skylake and Haswell, it does not fit as well on the Epyc processor. On this system, each socket actually embeds four NUMA places and would require one process per place. Third, the allocation strategy also needs to be performed manually on a per machine basis. Indeed, on the KNL machine, having one process per quadrant is not sufficient to reach top performance. Figure 2 shows that the MPI strategy equals or is better than the default strategy. However, the choice of memory kind clearly impacts the performance even more. The interleave strategy outperforms the MPI strategy, but AML remains on top for the fastest grids (*cf.* hash and unionized grid performance), because it prioritizes allocations on available MCDRAM memory.

To summarize, locality management of the XSBench proxy application is critical in order to attain good performance. Performance improvements can reach up to one order of magnitude when locality management is done correctly. Moreover, this performance can be achieved with little change to the application and few runtime tweaks. However, this method cannot be performed in a portable way across machines—a situation that will become increasingly harder to manage since future exascale systems are expected to feature deep and heterogeneous memory hierarchies. We therefore designed in AML an interface for replicating data on fast memories and perform local accesses in a portable way across architectures. Our addition to XSBench to use AML achieves best case performance in nearly all tested scenarios without changing the code, the number of processes, or the allocation strategy across machines.

3. Related Work

Data placement has been a topic of study since NUMA topologies became available, and continue to be critical as HPC systems grow in complexity. The PADAL whitepaper [11] recently identified locality-preserving abstractions in particular as a requirement to facilitate the development of HPC applications on future systems. Several frameworks have offered to simplify APIs for the allocation of data on NUMA topologies: the Simplified Interface for Complex Memory [12] or Hexe [13], using intents hints from the application programmer. Unfortunately these frameworks are not able to duplicate data to offer a different placement for each worker thread, nor taking into account actual performance measurements in the choice of device for an allocation. Interleave page allocation strategies like the ones present in the Linux kernel as also not ideal for such cases.

4. Conclusion

Monte Carlo neutron transport reactor simulation aims at acquiring data about the distribution and generation rates of neutrons within a nuclear reactor. High-performance simulation of a full-scale reactor requires assembling up to 1 TB of cross-section data. The random memory access and large memory footprint of the simulation make memory latency a major bottleneck on contemporary computing systems.

As we approach the exascale era, computing systems become more complex and feature deeper hierarchy of heterogeneous memories. Managing memory on such systems is increasingly hard. Embedding platform-specific ad hoc optimizations inside applications makes them nonportable and greatly increases the code complexity. In order to avoid this burden, a memory library called AML has been developed. AML abstracts memory management features with a consistent low-level interface across system devices.

Most representative computational hot spots of MC transport applications have been implemented in XSBench and RSBench miniapplications with state-of-the-art computational methods. The significantly simpler application structure of proxy applications enables us to exercise various optimizations on several computing systems to improve data access latency. To achieve this goal, we used the *area* abstraction of AML to implement localized memory allocators. On top of the areas we implemented a *replicaset* feature to copy data on low-latency memories. Using the hwloc library, we were able to match threads locations with a replica location and provide the application with local data on fast memories in a portable way. While adding only 4% additions to a version without AML, our optimization achieved to 2x speedup on contemporary Intel multisocket processors compared with a version without memory management. When moving to different machines, our optimizations consistently improved the XSBench figure of merit over low-effort optimizations.

Acknowledgements

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The material was based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

REFERENCES

[1] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz. "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis." In *PHYSOR 2014 - The International Conference on Physics of Reactors: The Role of Reactor Physics toward a Sustainable Future* (2014).

[2] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, and B. Forget. "OpenMC: A state-of-the-art Monte Carlo code for research and development." *Ann Nucl Energy*, **volume 82**, pp. 90–97 (2015).

[3] J. R. Tramm, A. R. Siegel, B. Forget, and C. Josey. "Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations." In *EASC 2014 - International Conference on Exascale Applications and Software: Solving Software Challenges for Exascale* (2014).

[4] J. Leppänen. "Two practical methods for unionized energy grid construction in continuous-energy Monte Carlo neutron transport calculation." *Annals of Nuclear Energy*, **volume 36**(7), pp. 878–885 (2009).

[5] F. Brown. "New hash-based energy lookup algorithm for Monte Carlo codes." *Transactions of the American Nuclear Society*, **volume 111**, pp. 659–662 (2014).

[6] C. Josey, P. Ducru, B. Forget, and K. Smith. "Windowed multipole for cross section Doppler broadening." *Journal of Computational Physics*, **volume 307**, pp. 715–727 (2016).

[7] A. Siegel, K. Smith, P. Fischer, and V. Mahadevan. "Analysis of communication costs for domain decomposed Monte Carlo methods in nuclear reactor analysis." *Journal of Computational Physics*, **volume 231**(8), pp. 3119–3125 (2012).

[8] K. Yoshii, J. Tramm, A. Siegel, and P. Beckman. "Improving the scalabiliy of neutron cross-section lookup codes on multicore NUMA system." *arXiv e-prints* (2019).

[9] S. Perarnau, B. Videau, N. Denoyelle, F. Monna, K. Iskra, and P. Beckman. "Explicit Data Layout Management for Autotuning Exploration on Complex Memory Topologies." In *Workshop on Memory Centric High Performance Computing (MCHPC)* (2019).

[10] B. Goglin. "Exposing the Locality of Heterogeneous Memory Architectures to HPC Applications." In *1st ACM International Symposium on Memory Systems (MEMSYS16)* (2016).

[11] D. Unat, J. Shalf, T. Hoefler, T. Schulthess, A. Dubey, and others (Eds.). "Programming Abstractions for Data Locality." Technical report (2014).

[12] "Simplified Interface to Complex Memory." https://github.com/lanl/SICM (2017).

[13] L. Oden and P. Balaji. "Hexe: A Toolkit for Heterogeneous Memory Management." In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)* (2017).